



NRL/MR/5540--04-8804

## A Snapshot View of SANE

MARGERY Y. LI

AMITABH KHASHNOBISH

JUDITH N. FROSCHER

*Center for High Assurance Computer Systems  
Information Technology Division*

June 11, 2004

Approved for public release; distribution is unlimited.

20040903 090



## CONTENTS

1	INTRODUCTION .....	1
2	UNDERLYING METHODOLOGY OF ASSURANCE ARGUMENT MAPS .....	2
2.1	The Assurance Strategy .....	2
2.2	Assertions and Assumptions .....	3
2.3	LAAMA Notation .....	5
3	FEATURES AND CAPABILITIES OF SANE .....	6
3.1	The Need for a Graphical Tool.....	6
3.2	Functional View of SANE .....	7
3.3	Implementation Details .....	9
4	TUTORIAL .....	10
4.1	System Requirements .....	10
4.2	Install SANE .....	10
4.3	Start SANE .....	11
4.4	SANE Views .....	12
4.5	Change the Look and Feel and System Configuration in SANE .....	13
4.6	Add/Delete Nodes in the SANE Editor .....	13
4.7	Modify a Node Attribute .....	14
4.8	Utilize the Integrated Dictionary for Claim Description .....	14
4.9	Support an Assumption/Evidence Using a Claim .....	15
4.10	Print an Assurance Argument Map .....	17
4.11	Save an Assurance Argument Map .....	17
5	PLANNED ENHANCEMENTS .....	18
	REFERENCE .....	18

# A Snapshot View of SANE

## 1 Introduction

Comprehensive certification of an information system in its operational environment challenges the most seasoned evaluators. In the early '90's, the Naval Research Laboratory introduced a comprehensive methodology for evaluating an operational system within its environment based on the assurance strategy. The assurance strategy or assurance map is a roadmap or an outline of a long, complicated assurance argument, which includes many kinds of evidence (e.g., security models, various system specifications, test results, covert channel analyses, vulnerability assessments, personnel clearances, physical constraints, procedural policies, etc.) for convincing the certifier and ultimately the accreditor that the risk of operating the system is low when compared to the benefits of using the system.

Certification differs from product evaluation in a very fundamental way. A vendor may expend significant resources in producing evidence to satisfy a given level of assurance for a product and may spend years attaining that level with the expectation of selling many copies of the product. On the other hand, a certification is a risk assessment of a system in its operational environment. The costs for certification are line items of the system acquisition cost. The system developer does not have the luxury of repeated attempts to satisfy a criteria but must instead, identify the vulnerabilities in the system and mitigate them with other cost effective mechanisms. Some of these mechanisms will be restrictions about who can use the system and some will constrain the operational environment of the system. Understanding how all these mechanisms collectively protect sensitive information is a fundamental aspect of certification. It is important to note that an evaluated product will only process sensitive data when it becomes a component of a certified system. The NRL methodology attempts to provide a systematic, comprehensive approach for making decisions about what mechanisms to use, how a mechanism depends on others, how a vulnerability can weaken the protection posture of the whole system, etc. and how changes to an already certified system impact the overall security posture.

SANE (Security Assurance Navigation Environment) is a tool for developing an assurance argument map for a system to be certified in its operational environment. The assurance argument map is used as a roadmap for evaluating different security components (IT and non-IT) of the system where mission-critical information may be compromised. Using the assurance argument map during the certification process facilitates identification of security risks. It can also suggest optimal countermeasures to these risks. SANE provides a comprehensive set of tools to construct assurance argument maps that help certifiers evaluate assurance components of a system and that enable system designers and developers to make informed trade-offs about protection software. Finally, it can be used as a communication medium between system developers and certifiers when the latter are involved early in the development process.

In this paper, we give a snapshot view of SANE. Section 2 briefly discusses the underlying methodology and concepts behind assurance argument maps. Users wanting greater detail can refer to [1][2]. Section 3 presents SANE's features and capabilities. Section 4 provides a tutorial on using SANE. Section 5 concludes with a discussion of planned future enhancements.

## 2 Underlying Methodology of Assurance Argument Maps

Ensuring that a given system meets its security requirements is not an easy task. Traditional methods of evaluations and certifications provide a piecemeal approach to risk assessment. Although these methods do provide ways of assessing a system in specific environments with specific threat levels, they do not identify all the risks of operating the system in its operational environment, nor do they offer trade-off functions to mitigate such risks [3]. What is needed is a comprehensive approach that can provide a means for reasoning about all the protection mechanisms needed for securing an operational system and for understanding the dependencies among the mechanisms. There is a desire to use the Common Criteria to evaluate an operational system [4]. However, the Common Criteria provides an approach for *product evaluation* in isolation and not the totality of its operation within a system. The *system certification* process is a superset of product evaluation, and is typically much more comprehensive. There are a great many other risks and potential threats that arise when viewing a system in its operational environment. The Common Criteria uses a textual representation methodology similar to the NRL approach to record the security claims and the vulnerabilities of each product in a Protection Profile; SANE provides a graphical depiction of the same information.

Identifying the potential risks within the operational environment in which a system operates involves assessing different security disciplines and the dependencies among them. Evaluating the IT components, the software and the hardware, alone will not suffice because they are not the only components that make up the operational security environment as a whole. Other non-IT security disciplines, such as the people that use the system, the physical environment in which the system resides and the administrative measures that regulate the usage of the system must also be evaluated. Incorporating non-IT security disciplines into the assessment process is crucial because they can compensate for uncertainties or weaknesses and offer countermeasures, which IT cannot provide. When all these security disciplines come into play, we must apply a strategy that can evaluate each security discipline against its assurance criteria, analyze the relationships and conflicts among the disciplines and compute risk factors based on the information collected above. Once the risk analysis is generated, system developers can compare it against the system requirements and make trade-off decisions to divert and mitigate any potential risks within the system.

### 2.1 The Assurance Strategy

The strategy that we use to perform the risk analysis is called the assurance strategy. It was introduced by Payne [3] who created the strategy framework by

extending the attributes of Informal Security Model proposed by Landwehr [5] and Froscher [6]. The assurance strategy helps the system developers identify relationships among various security disciplines by documenting the trade-off decisions based on a set of assertions and assumptions that must be true for the system as a whole to be secure. It starts out identifying the system's information security (INFOSEC) policy. As illustrated in Figure 1, the INFOSEC policy is derived from the organizational security policy and from other objectives and constraints [3]. With the INFOSEC policy in place and the operational requirements defined, the assurance strategy further categorizes the trusted system's security requirements into four primary security disciplines using DOD's Network Rating Methodology (NRM) [1] and constructs an assurance argument map that consists of a set of assurance arguments derived from each security discipline.

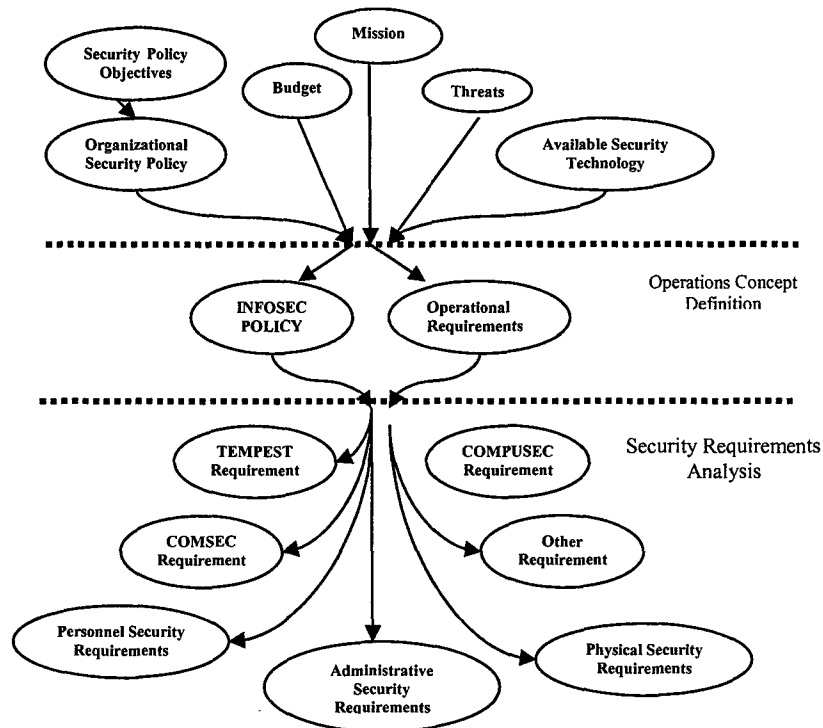


Figure 1 – INFOSEC Policy

## 2.2 Assertions and Assumptions

An assurance argument map is a representation of the assurance strategy. It depicts an overall view of all the security requirements that must be met for a system to be secure within its operational environment. The map is constructed using a set of assurance arguments. Each assurance argument is a complex chain of reasoning that justifies a claim made about a particular area of security discipline within a system. There are four primary security disciplines defined by the NRM approach, a simplification of the NRL certification methodology. They are Physical Security, Technological Security, Operational Security and Personnel Security (Figure 2).

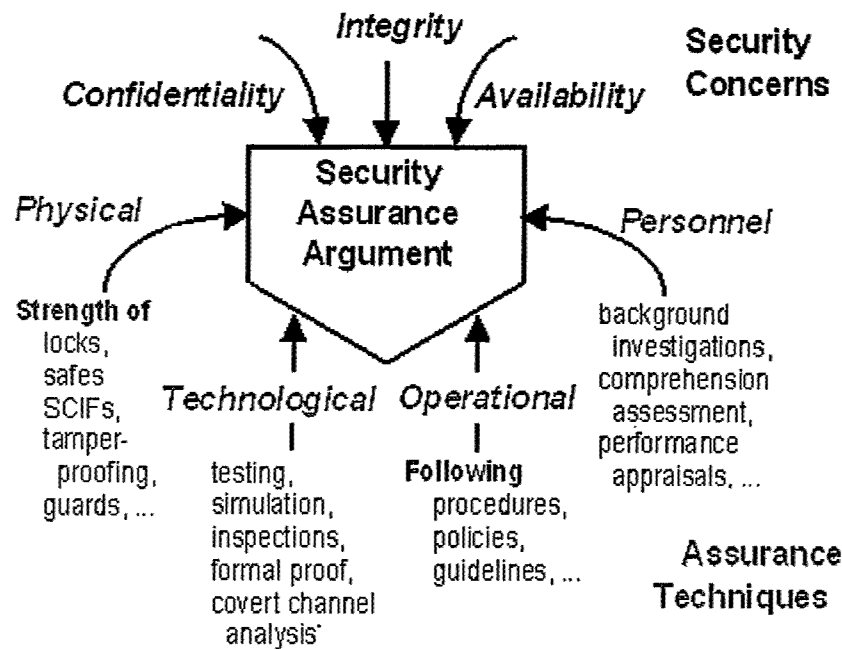


Figure 2 – NRM Security Assurance Arguments

A claim can be in the form of an assertion or an assumption. An example of a Technological Security claim is

*The safehost device in a host will return a unique authenticated check indicating whether the host is safe or unsafe when a secure attention instruction is performed.*


This claim asserts that the safehost device will perform a check when it is asked to. However, it does not assert that the safehost device has been installed correctly. Therefore, to support this claim, an assumption must be made


*The safehost device is properly installed.*


From the computer security point of view, this assumption cannot be enforced by the computer software and hardware. However, from the standpoint of operational security, it is an assertion [3]. Clearly, security assumptions in one security discipline can be mapped as security assertions in another security discipline; a gap in this mapping indicates a vulnerability [7]. Using assertions and assumptions as claims in the assurance argument map facilitates the accreditation process. But we need to find a way to present different claims in an assurance argument map. For this purpose, a graphical language is introduced into the framework. It is called the Language of Assurance Argument Maps (LAAMA).

## 2.3 LAAMA Notation

LAAMA is a simplified version of the Composite Assurance Mapping Language (CAML). CAML is the first generation graphical language used in building assurance argument maps [2]. It is set of distinct graphical primitives that come in ten different shapes. Each primitive represents a key component of the argument map. A textual summary of each component is shown inside each primitive shape. Although CAML offers many different primitive shapes, it has proven to be quite complex to use. Often time, users have difficulty in associating each primitive shape with its semantics. Also, an argument map with too many different shapes of primitives becomes harder for the user to trace the argument logic. LAAMA is a refinement of CAML. It extracts the core graphical components of CAML to form a language that is compact and easy to use. The following are the primitive shapes in the LAAMA notation.

 - Claim nodes (AND, OR decomposition) {**BLUE**}. Claims can be refined into (validated by) lower level sub-claims, assumptions and evidence. Claim nodes with AND decomposition require all subnodes to be valid, while those with OR decomposition require just one of their subnodes to be valid.

 - Assumption node {**RED**}. An assumption is set to an invalid state by default and cannot be refined into lower level sub-claims. It can be validated by claims or evidence from a different security discipline or from a separate argument tree. Once validated, the color of the assumption will turn green indicating its validity. Invalid assumptions that lack support from claims or evidence indicate vulnerabilities to threats.

 - Evidence node {**PERIWINKLE**}. Like an assumption, it cannot be refined into lower level sub-claims. Its role is to support assumptions and claims.

With the LAAMA notation in place, each assurance map has the following properties

- *Claims* must be proven for each discipline and can be refined into lower level claims that demonstrate their validity.
- *Evidence* that demonstrates the validity of a claim is linked to that claim.
- *Assumptions* represent those claims about a system's or a component's environment that need be true for the associated claim to be valid. Assumptions themselves need to be independently validated.
- *Claims* from one part of the map, a sub-map, can validate assumptions from another. Invalidated assumptions represent vulnerabilities in the system

An argument map exhibits a set of tree-like behaviors. It starts with a set of high level claims or assumptions, derived from the system's security objectives, and works its way down by decomposing and refining each claim. During the process of refinement, a sub-map from one security discipline and/or a sub-map that represents a component may be added to the map. This process continues for all defined security disciplines and security components. This approach helps the system developers organize a set of



assurance arguments into a structured flow that is easy to follow and analyze. They can use this map to look at each security discipline, study the relationships among them, mitigate possible threats and vulnerabilities in the system with countermeasures and make informed trade-off decisions. In addition, the map can be used as a communication medium between the certifiers and system developers. Getting certifiers involved early during the development phase is crucial. They can help identify erroneous and costly decisions developers might make during the development process before it is too late. The map also provides an understandable technical context for program managers and approval authorities to make informed risk management decisions.

### **3 Features and Capabilities of SANE**

#### **3.1 The need for a graphical tool**

The Assurance Argument Map methodology benefits greatly from the use of an appropriate graphical tool. A well decomposed argument map can get large, unwieldy and error-prone. There are many compelling arguments for the use of a graphical tool.

- A graphical tool greatly eases navigating large complex trees
- A real-life argument map will span different security disciplines and even different domains within each discipline, requiring different domain experts. Development of such a map is a collaborative effort. A tool with client-server components can greatly facilitate such collaboration
- A real-life argument map involving different domain experts can also include sensitive information including vulnerabilities that should not be disseminated widely. As such, a tool that supports access-controls and enforces a need-to-know policy can facilitate collaborative efforts without giving everyone access to everything.
- The process of developing a complex argument map can be lengthy, involve various wrong turns, and possibly developer turnover. In such a scenario, a tool that efficiently tracks the development trajectory can be useful, allowing developers to get a sense of the development history, back-track out of wrong development paths, branch out on alternative paths, merge such alternative paths and provide general audit information to track tradeoff decisions made.
- A graphical tool can make it easy to reuse previously analyzed and decomposed argument-subtrees. Very often, there are common arguments that get used over and over on different higher-level claims. A graphical tool, accompanied by an argument repository can facilitate such reuse.
- A graphical tool can automate the process of verifying validity and the enforcement of constraints. Verifying the validity of a given node can be fairly involved, and it is often difficult for the developer of an argument to quickly check whether a given node is valid. Also, a valid argument map has to follow various constraints. A tool can make it easy to avoid violating these constraints during the development process.

- The development of an argument map requires using a terminology database. It is important to be exact and consistent with terms. The developer should avoid using the same term with different semantics or use different terms to convey the same meaning. An integrated dictionary tool can greatly aid the developer in this respect, by keeping the definition of terms just a click away.
- The ultimate purpose of an argument map is to make a structured assurance argument. However, presenting the raw (sometimes complex) argument map overwhelms the assessor with too much data. A tool is needed that can allow the presenter to selectively display information and markup the tree in a more presentation-friendly fashion.

### 3.2 Functional view of SANE

The SANE frontend has at its core a tree editor. An assurance map is presented as a tree, with nodes representing either claims, assumptions or evidence. The user can add, modify or delete nodes subject to some basic structural constraints, i.e. only claims may have subnodes. Each node, depending on its type, has various attributes that the user can change, subject to access control restrictions (more about that later).

All actions within SANE are reversible, i.e. the user can undo and redo actions. All actions within a session are undoable without limits. SANE maintains an execution log of all transactions that occur within any given session.

The user environment within SANE has customization and ease-of-use features. All system configuration settings are accessible from the GUI. All user actions can be placed on the toolbar. Most user actions are accessible from a context-sensitive right-click popup menu in addition to being available from the toolbar or menubar. Most menu actions are accessible via keyboard shortcuts, a valuable plus for power-users.

The GUI allows the user to load and save an argument map to a remote repository, in addition to allowing them to be saved to the local filesystem. In addition, while loading from the repository, the user has easy access to all previous versions of the map that have been committed to the repository during the development process. This makes it easy to back out of some wrong turns taken during the development process.

SANE has access-control capability for argument maps. The purpose of access-control is three-fold

- It enforces need-to-know for portions of the tree that cover sensitive information.
- It allows different domain experts to work on different portions of the assurance map without stepping on each-others toes. It formalizes the areas of responsibility.
- It allows for users with different *roles* to work on overlapping portions of the map.

Access control is implemented on a per-node basis. By default, a node within the tree belongs to the user that created it. The user is then the owner of that node and all the

subnodes that she subsequently creates. When the owner decomposes a node into subnodes that fall outside her area of expertise, she can delegate those nodes to the appropriate domain expert. She does that by transferring ownership of those subnodes.

In addition to ownership, a node has an access control list of authorized users and their role for that node. This list can only be modified by the owner. Each role has an associated access-right for each attribute within the node. The access rights are NOACCESS, READ and READ-WRITE. There are three predefined roles: Developer, Certifier and ReadOnly. The owner is automatically a developer for that node. Certifiers have read-write access to certifier-specific fields within the node and read-only access to all other fields. Developer access is the complement of certifier access.

User identification is based on certificates (more on that later). Access-control can only be enforced in the context of using the repository.

SANE comes with an integrated dictionary. The dictionary allows users to precisely define terms used within the argument. All appearances of such terms are automatically hyperlinked to their definition within the dictionary. Also, it is possible to track all usages of a defined term within a particular argument tree.

SANE comes with version control facility, implemented on the server-side repository. The server uses an underlying CVS engine.

SANE simplifies the process of delegation. When a developer starts to decompose an argument, she will often come upon an area where she lacks expertise. At this point, she can simply assign that subnode to the appropriate expert. In a collaborative scenario, developer's checking-out an argument tree using the GUI frontend will quickly be able to identify the nodes that are their responsibility. They can then either validate them or decompose them further, and in the process again delegate subnodes to some other experts.

At any point in the decomposition of an argument, if the user comes across an argument that she thinks might have been previously validated using the argument-map approach, she can search the repository and import the requisite subtree. This greatly facilitates Reusability.

A node is only valid if both its own decomposition and the entire subtree, down to the last nodes, are valid. SANE automatically computes the validity of a node and highlights invalid nodes. Assumptions are usually validated by claims in other trees. SANE makes it easy to create this link, and it computes the validity of the assumption by computing the validity of the external linked claim.

SANE generates a postscript-based image of the tree, which is less compact but more visually appealing. This also makes it easy to interface the generated charts with other applications using the platform-neutral postscript format.

SANE has selective display capability, allowing the viewer to view the argument map at any level of abstraction. Users can toggle the sub-tree view of any node within the tree. Viewers can also zoom and pan over large trees.

SANE recognizes the *user-id* of the user. It then maps the id into a *role* for each of the nodes. The role decides the level of access the user has to modify the argument tree. The important roles are *developer* and *certifier*. These roles constrain the user in important ways. The certifier can validate a node or tack on certifier comments. The developer can change most other attributes of the node. The developer can also set different access policy based on the role for the development team. This capability allows different people to work in different area of expertise.

SANE allows the user to view the argument tree document in many different ways. It has a form-view, postscript-view, xml-view and dot-view. Each of these views exist on different tab-panes within the application.

SANE comes with extensive integrated context-sensitive help

### 3.3 Implementation details

SANE is implemented as a client-server application. Most of the important functionalities of the application are implemented on the client-side as a GUI. The client enables the user to perform the basic editing functions on the argument map. The more advanced functionalities: collaboration (with associated access-control and need-to-know requirements), version-control, reuse etc. require the server-side.

The client is implemented in Java using the swing library. It is thus cross-platform. It utilizes several third-party components

- The rendering of the graph uses *Graphviz*, an open-source graphing utility.
- The postscript rendering is done using a free postscript rendering utility.

The server is implemented using Java. It utilizes CVS as a back-end to provide the version-control capability. The server also does pre and post-filtering while servicing client requests in order to enforce access-controls. After authenticating the client, the server determines the clients role for each node within the argument-tree being served. During a checkout, the server will remove all nodes to which the client lacks read access. During a checkin, the server will filter out all nodes for which the client lacks write access and retain the values from its original copy.

User identity and middleware: Communication between client and server utilizes RMI over SSL. SANE uses the *Java Cryptographic Extension (JCE)* and certificates to authenticate the user to the server. All invocations between client and server utilize RMI running over SSL. Each client invocation is authenticated by server before proceeding.

When a user launches the SANE client, the client will prompt the user for a password to unlock its identity from the local key-store. The client will then use this identity for all subsequent communication with the server. Additionally, the client also ships with the certificate of the server. Thus all subsequent communication between client and server is bi-directionally authenticated and encrypted.

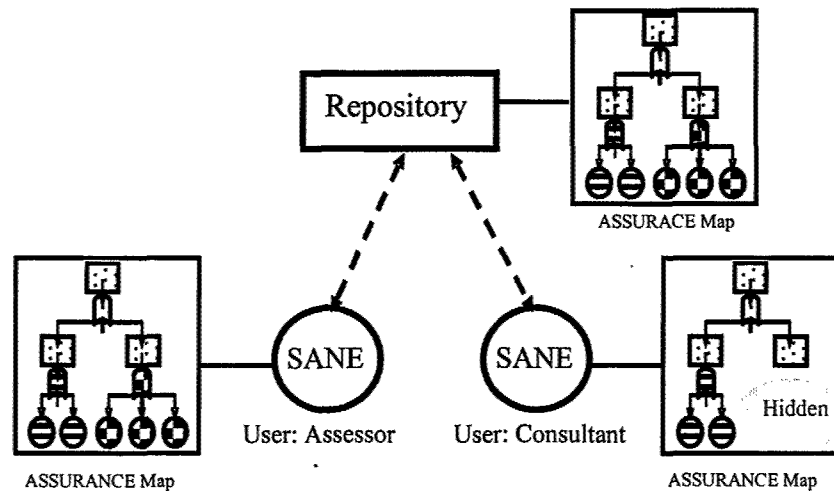


Figure 3 - Information Sharing and Hiding by SANE

## 4 Tutorial

### 4.1 System Requirements

The following are the minimum requirements for running SANE:

Operating Systems: Windows 2000, XP or Linux operating system  
Hardware: Pentium-class personal computer  
Memory: 32 megabytes of RAM or more  
Java Runtime Environment: JRE Std. Ed. v1.4.0

### 4.2 Install SANE

There are two tarballs needed for the SANE installation. Besides its own system libraries (which will come bundled in a tarball for download), SANE makes use of a graphical tool, Graphviz, to render the argument map in the postscript view. It is open source licensed software by AT & T Labs. To download Graphviz, please visit:

<http://www.research.att.com/sw/tools/graphviz/download.html>

After both tarballs are obtained, move the files into a designated\_directory before extracting them. To extract the contents of the tarballs in a Windows environment, one

can use winzip or a default tool that comes with Windows. For those who use Linux, open a command prompt, locate the designated directory and execute the following command:

```
username/home/designated_directory: tar xvfz sane.tar.gz graphviz.tar.gz
```

Once both tarballs are decompressed, please read the README files for installation instructions.

### 4.3 Start SANE

To start the tree editor (Figure 4) in the SANE application, open a command prompt in Windows or Linux and locate the designated directory where SANE resides and execute the following command:

Starting in Windows:

```
c:\designated_directory> cd scripts  
c:\designated_directory\scripts> run
```

Starting in Linux:

```
username/home/designated_directory: cd scripts  
username/home/designated_directory/scripts: ./run
```

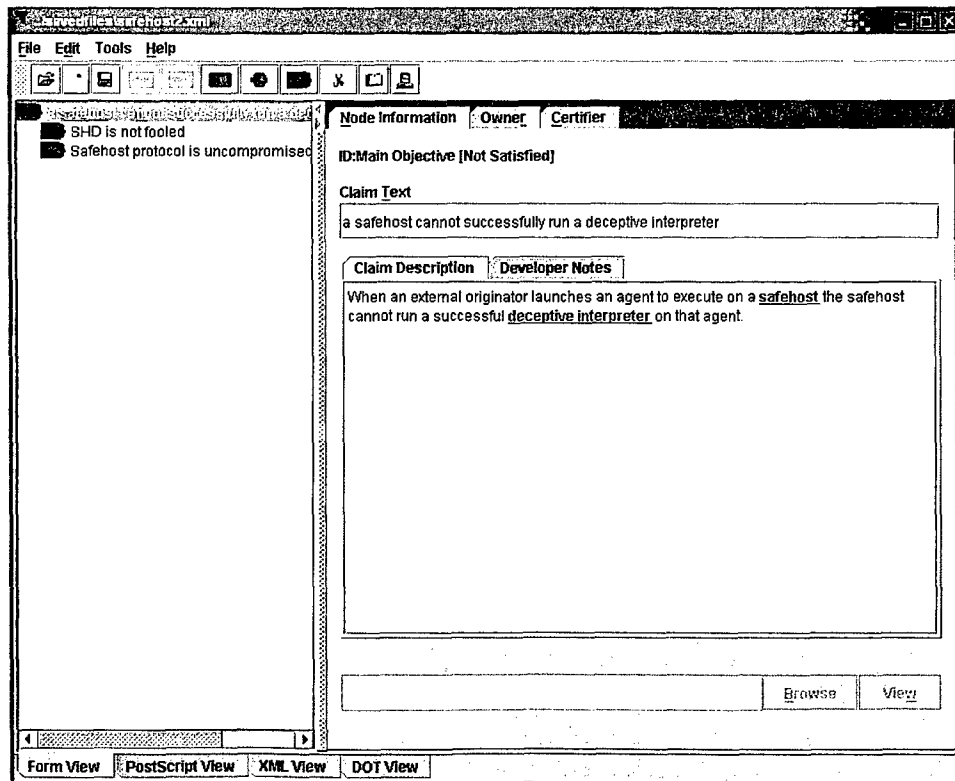


Figure 4 – SANE in Form View

## 4.4 SANE Views

SANE offers two different views of an argument map. By default, the Form View will be displayed when the application starts (Figure 4). The Form View offers two panels. The left panel displays the argument map in a hierarchical tree structure and the right panel shows detailed information of the node that is being highlighted on the left panel in various form fields.

The PostScript View displays the argument map in a more graphical view (Figure 5). But unlike the Form View, it is not intended to be used when building the argument map. XML view and DOT view provide in-depth information on how the argument map is constructed. XML (Extensible Markup Language) is used to store structured data in a text format. Data stored in such format can be easily retrieved and modified. DOT is a preprocessor used by Graphviz for drawing directed graphs. For those wanting more information on DOT, please refer to [8]. Finally the Execution Log documents each transaction the user made and allows her to backtrack.

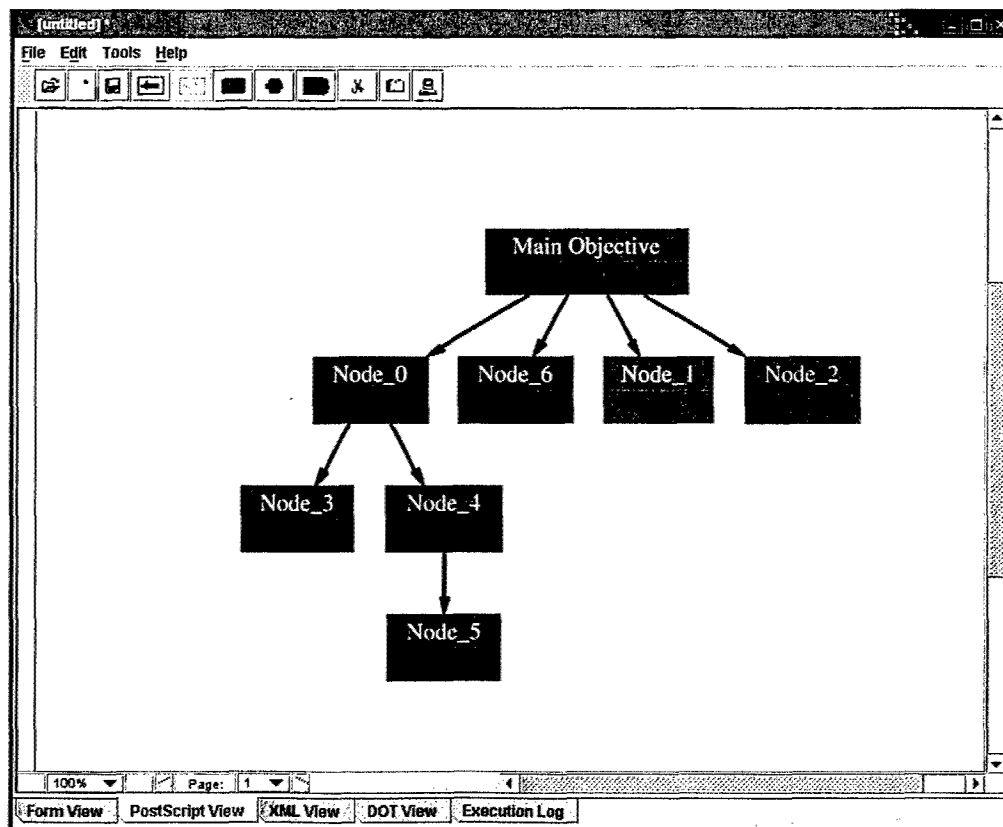


Figure 5- SANE in PostScript View

## 4.5 Change the Look and Feel and System Configuration in SANE

SANE allows users to customize their own look and feel and modify the system configuration. To customize the tool bar, go to the menu bar and click on *Tools* and then *Edit ToolBar*. A toolbar editor will come up allowing users to pick and choose which tools they want in the toolbar (Figure 6).

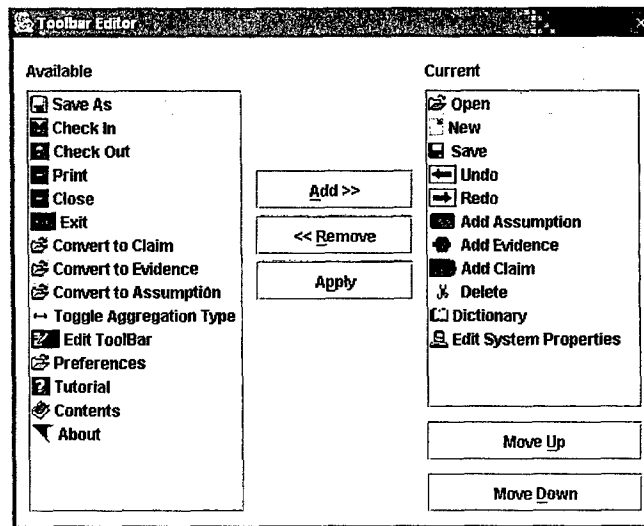


Figure 6 – SANE Toolbar Editor

To modify the system configuration, click on *Edit System Properties* under *Tools*. A system-property display will come up allowing users to make modification to it.

## 4.6 Add/Delete Nodes in the SANE Editor

According to the basic structure constraints, only claim nodes may have subnodes. In addition, each claim node is a conjunction of its subnodes by default. This can be changed to a disjunction using the *Toggle Aggregation Type* in the menu item. Typically, disjunctions are used only at the abstract level (in the main objective) of an assurance argument map whereas conjunctions are applied when doing claim refinement. This makes sense since refinement means decomposing a claim into a set of subclaims. There are three ways to add or delete a subnode to a claim.

- Use the icons on the tool bar
- Use the “Edit” menu on the menubar
- Use the context-sensitive right-click popup menu (Figure 7)



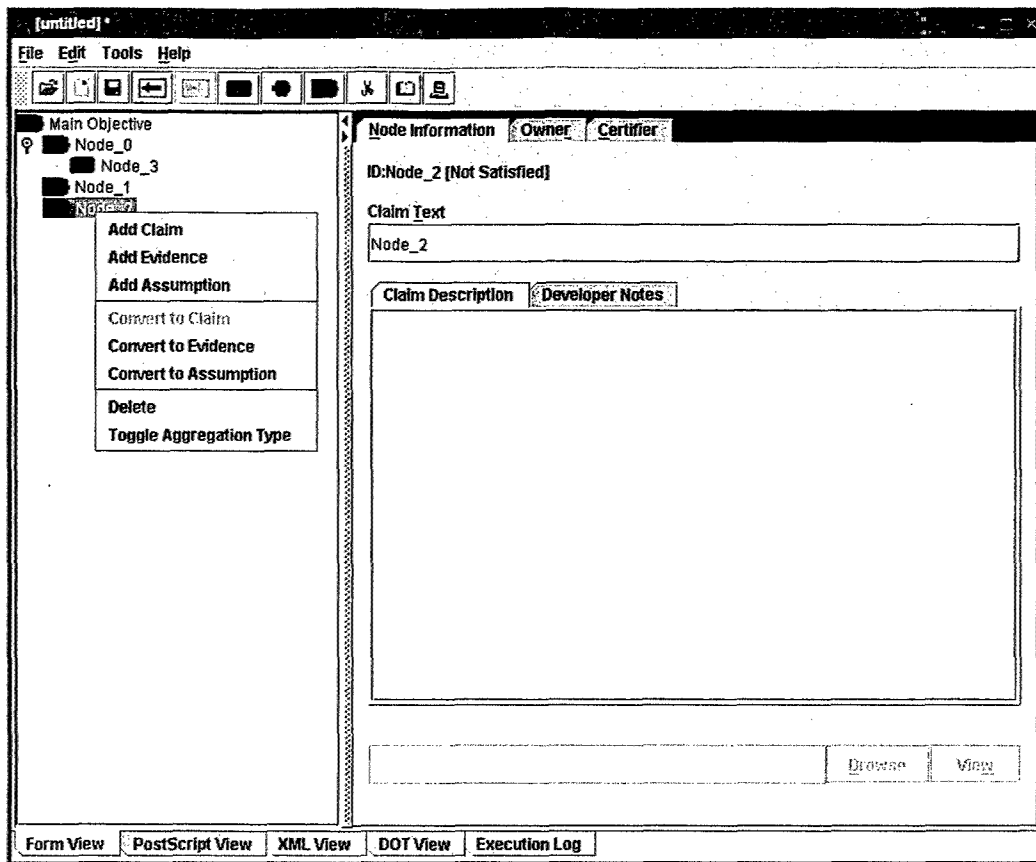


Figure 7 – SANE Popup Menu

#### 4.7 Modify a Node Attribute

Each node has a Node Information panel associated with it. The panel describes all attributes of that node. All the fields in the panel can be changed except the node ID. To modify an attribute, simply point the cursor of the mouse to the textfield of that attribute and type.

#### 4.8 Utilize the Integrated Dictionary for Claim Description

The dictionary stores predefined terms within the argument. When the user describes a claim within its claim description, all appearances of these terms are hyperlinked to their definition in the dictionary (Figure 4). Each defined term appears in red and is underlined. To define new terms, click on the dictionary icon on the menubar to bring up the dictionary (Figure 8).

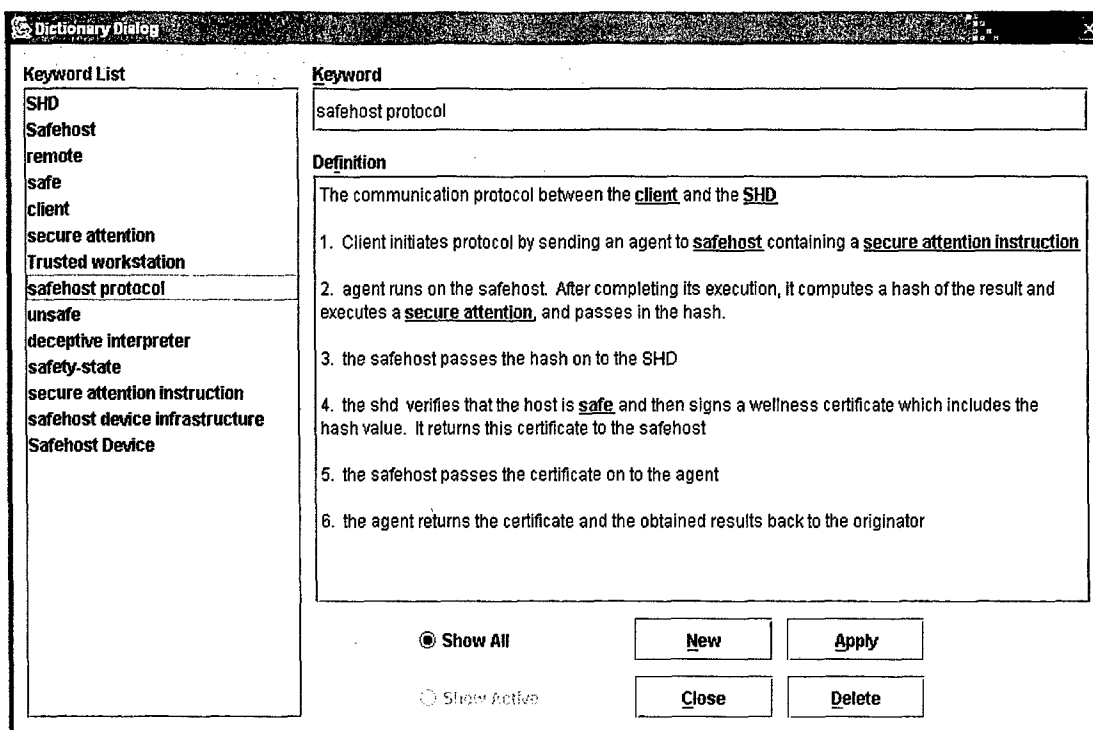


Figure 8 – SANE INTEGRATED DICTIONARY

#### 4.9 Support an Assumption/Evidence Using a Claim

A claim can be used to support an assumption or evidence. To support an assumption, do the following:

- Highlight the assumption node on the left panel of the Form View.
- Click on the Browse button next to the textfield that labeled “URL of Assumption” in the Node Information panel.
- A system file dialog will come up allowing the user to choose the target file where the supporting claim is.
- Click on the name of that file to open and a new SANE editor will come up displaying the argument map of the selected file.
- Right-click on the desired claim to bring up the popup menu. Notice that all other menu items are grayed out except *Set As Assumption Target* (Figure 9). In addition, the new editor has the title *Pick Target By Right-Clicking on Node* located on the upper left hand corner. Setting the assumption target of the desired claim will cause the new editor to close and set the correct url in the textfield of the assumption node (Figure 10).

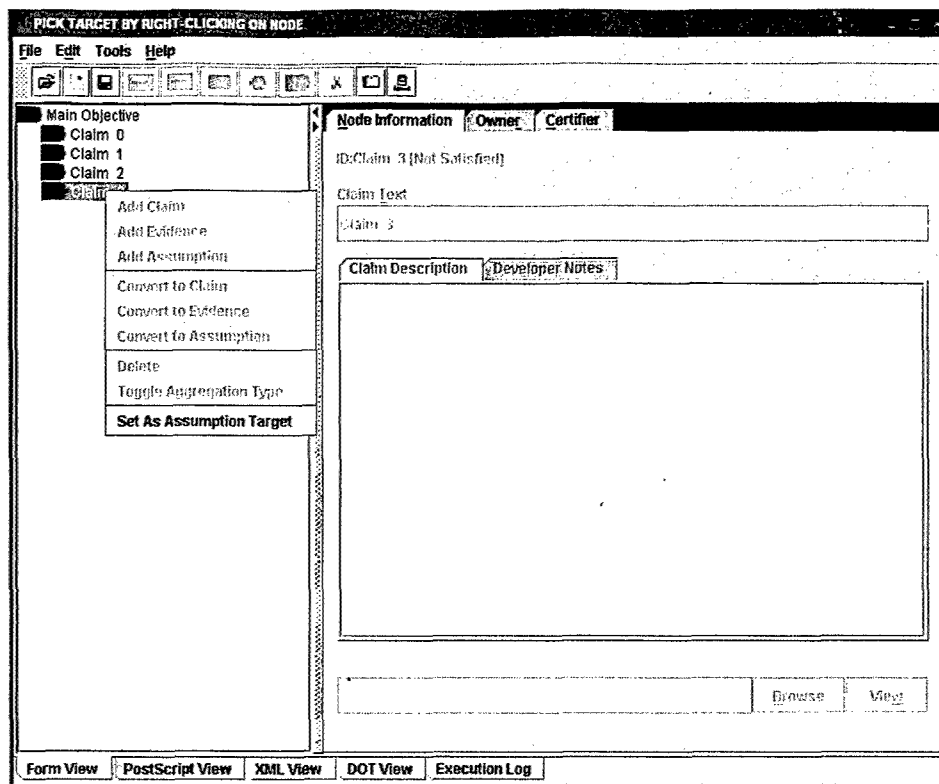


Figure 9 – Set Assumption Target

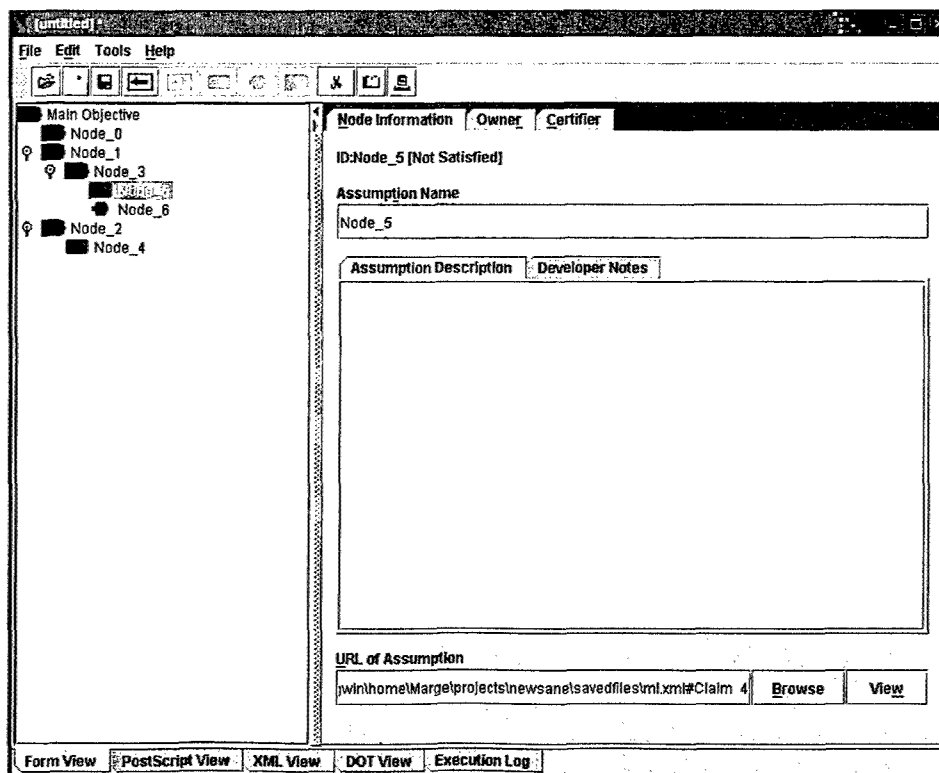


Figure 10 – An Assumption That Is Supported by a Claim

## 4.10 Print an Assurance Argument Map

To print an assurance argument map in SANE, click on the *printer* button on the panel on the menu item and the print job will be sent to the user's default printer.

## 4.11 Save an Assurance Argument Map

An assurance argument map will be saved in XML. It can be saved in the user's local file system or be checked in to a remote repository. To save it locally, simply click the *save* button on the menu item and the system file dialog will come up. Select the desired subdirectory in which the file will be saved. Enter the document name and save it with a .xml extension. Once the assurance argument map is saved, the title of the SANE editor will reflect its file name. Subsequent saves will not prompt for a file name.

To save it remotely, click on the *check in* button on the menu item and a Checkin Form will appear (Figure 10). To retrieve from the remote server, click on the *check out* button. The checkout form allows user to retrieve the argument map, as well as all previous versions of it.

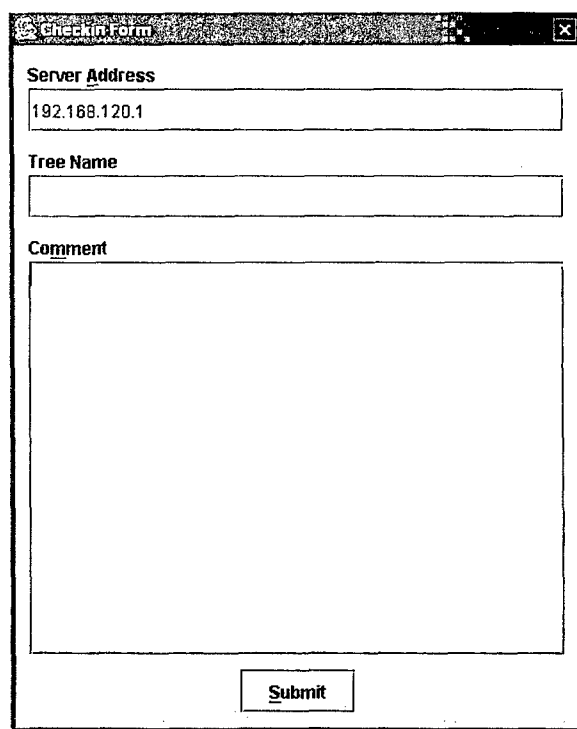
The image shows a screenshot of a 'Checkin Form' dialog box. The dialog has a title bar with the text 'Checkin Form' and a close button. Inside the dialog, there are three input fields: 'Server Address' with the value '192.168.120.1', 'Tree Name' which is empty, and 'Comment' which is a large text area. At the bottom right of the dialog is a 'Submit' button.

Figure 10 – SANE Checkin Form

## 5 Planned Enhancements

Enhancement will be made based on feedback from users using the tool. Currently, SANE is being used to map an assurance argument for the Programmable Embedded INFOSEC Product (PEIP) being developed by NRL under Navy funding. It is also being used to develop an assurance argument for SAFEHOST, which is being developed by NRL.

## REFERENCE:

- [1] NRM Authors Group, The Network Rating Model: A Methodology for Assessing Network Security  
<http://www.radium.ncsc.mil/nrm/rev961031.html> SECOND DRAFT, National Security Agency, 31 October 1996, (accessed on 12 February 2003).
- [2] J.S. Park, B. Montrose and J.N. Froscher, "Tools for Information Security Assurance Arguments", DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings, Volume 1, P. 287-296
- [3] C.N. Payne, J.N. Froscher and C.E. Landwehr, "Toward A Comprehensive INFOSEC Certification Methodology," Proc. Sixteenth National Computer Security Conference, Baltimore, MD, Sept., 1993. pp. 165-172.
- [4] Common Criteria Implementation Board, Common Criteria for Information Technology Security Evaluation, CCIB-98-026, August 1999.
- [5] C. Landwehr, C. Heitmeyer, and J. McLean, "A security model for military message systems," ACM Transactions on computer System, vol.2, pp. 198-222, August 1984.
- [6] J. Froscher and J. Carroll, "Security requirements of navy embedded computers," NRL Memorandum Report 5425, Naval Research Laboratory, September 1984.
- [7] A.P. Moore, J.E. Klinker and D.M. Mihelcic, "How to Construct Formal Arguments that Persuade Certifiers", chapter in "Industrial Strength Formal Methods in Practice" Springer Verlag London Limited, eds. M.G. Hinchey and J.P. Bowen, September 1999.
- [8] Graphviz- open source graph drawing software.  
<http://www.research.att.com/sw/tools/graphviz> (accessed on 15 December 2003).